Statistical learning (MT7038) - Project 1

Sergio Arnaud

5 december, 2019

Instructions: *This project consists of five tasks that should be solved individually. You are free to copy and modify code used in the project template file* ***MT7038_project1_VT19.Rmd*** *without further reference, but any code inspired by fellow students or other sources must be clearly acknowledged. Use of specific R-packages apart from those used in the template (the* ***tidyverse****-suite,* ***rpart*** *and* ***rpart.plot****) requires permission from course staff.*

The solution should be submitted at the course web page as a source markdown file together with a compiled pdf.

```
library(tidyverse) # For data manipulation
library(rpart) # For trees in later part
library(rpart.plot) # For trees in later part
library(gridExtra)
theme_set(theme_minimal()) # ggplot theme, set as you like
```

We start by reading the Prostate Cancer data described in section 3.2.1 of the textbook

```
# Import
url <- "https://web.stanford.edu/~hastie/ElemStatLearn/datasets/prostate.data"
prostate_raw <- read_tsv(url) %>% select(-X1)
```

Splitting in test and train datasets

```
# Split into separate tables
prostate_raw_train<- prostate_raw %>%
    filter(train) %>%
    select(-train)
prostate_raw_test <- prostate_raw %>%
    filter(!train) %>%
    select(-train)
```

We continue by - standardising variables - randomly reordering observations - setting a seed for reproducibility

```
set.seed(970628)

# Prepare by scaling and random reordering
col_mean <- map(prostate_raw_train, mean)
col_sd <- map(prostate_raw_train, sd)
prostate_prep_train <- prostate_raw_train %>%
    map2_df(col_mean, ~.x - .y) %>% # Remove mean
    map2_df(col_sd, ~.x / .y) %>% # Divide by sd
    slice(sample(1:n())) # Random reordering

prostate_prep_test <- prostate_raw_test %>%
    map2_df(col_mean, ~.x - .y) %>%
    map2_df(col_sd, ~.x / .y)
```

Task 1

In the above code, the mean and standard deviation of the training set was used to standardise the test set. Argue why this is a good idea.

There are two other ways in which we can standardise the train and test sets:

1) Using the global mean and standart deviation to standardise both the test and train datasets.
2) For each set, use it's own mean and standart deviation.

We will argue why both 1) and 2) are wrong.

The first case uses information from the test set and we could see that as *cheeting* since such information should never be used before the evaluation of the model. Indeed, we are *overusing* the test information and the consequence of that is that we underestimate the real prediction error as seen in the section 'the wrong and right way of do cross-validation' in the ESL.

The second case induces inmediatly a bias. Suppose that we have an observation in the train set and, when testing the model we happen to have the exact same observation.Since we are using different scalings they would have different values when feed to the model and, as a consequence, be treated differently by it.

Cross-validation

The function `cv_fold` below splits data into `n_fold` folds and places the corresponding train/test-sets in a tidy data frame.

```
cv_fold <- function(data, n_fold){
    # fold_id denotes in which fold the observation
    # belongs to the test set
    data <- mutate(data, fold_id = rep_len(1:n_fold, length.out = n()))
    # Two functions to split data into train and test sets
    cv_train <- function(fold, data){
        filter(data, fold_id != fold) %>%
            select(- fold_id)
    }
    cv_test <- function(fold, data){
        filter(data, fold_id == fold) %>%
            select(- fold_id)
    }
    # Folding
    tibble(fold = 1:n_fold) %>%
        mutate(train = map(fold, ~cv_train(.x, data)),
               test = map(fold, ~cv_test(.x, data)),
               fold = paste0("Fold", fold))
}
```

We now apply it to the training data using 10 folds, any further tuning should be based on the resulting `cv_prostate` data frame.

```
n_fold <- 10
cv_prostate <- cv_fold(prostate_prep_train, n_fold)
glimpse(cv_prostate)
```

```
## Observations: 10
## Variables: 3
## $ fold  <chr> "Fold1", "Fold2", "Fold3", "Fold4", "Fold5", "Fold6", "Fold7"...
```

```
## $ train <list> [<tbl_df[60 x 9]>, <tbl_df[60 x 9]>, <tbl_df[60 x 9]>, <tbl_...
## $ test  <list> [<tbl_df[7 x 9]>, <tbl_df[7 x 9]>, <tbl_df[7 x 9]>, <tbl_df[...
```

Fitting a ridge regression model

Task 2

Change the random seed to your date of birth (yymmdd). Rerun the analysis and compare the figure of mse versus lambda with the one in this sheet, you may need to adapt `lambda_seq` *in order to fit the minimum within the range of lambdas. Does the optimal lambda seem to be sensitive to the random splitting in folds? Plot the mse for each fold (rather than the average) in the same figure and report any conclusions.*

```
lm.ridge <- function(data, formula, lambda){
    # Given data, model formula and shrinkage parameter lambda,
    # this function returns a data.frame of estimated coefficients
    X <- model.matrix(formula, data) # Exctract design matrix X
    p <- ncol(X)
    y <- model.frame(formula, data) %>% # Extract vector of responses y
        model.extract("response")
    # Compute parameter estimates (Eq. (3.44) in textbook)
    R <- t(X) %*% X
    solve(R + lambda * diag(p)) %*% t(X) %*% as.matrix(y) %>%
        as.data.frame() %>%
        setNames("estimate") %>%
        rownames_to_column("variable")
}

# Function that predicts the respons
predict.ridge <- function(newdata, fit, formula){
    model.matrix(formula, data = newdata) %*% as.matrix(fit$estimate) %>%
        as.numeric()
}
```

We fit a model with all explanatory variables and without an intercept

```
formula_ridge <- lpsa ~ -1 + .
```

With the following sequence of hyperparameters

```
lambda_seq <- exp(seq(-.5, log(10), length.out = 20))
```

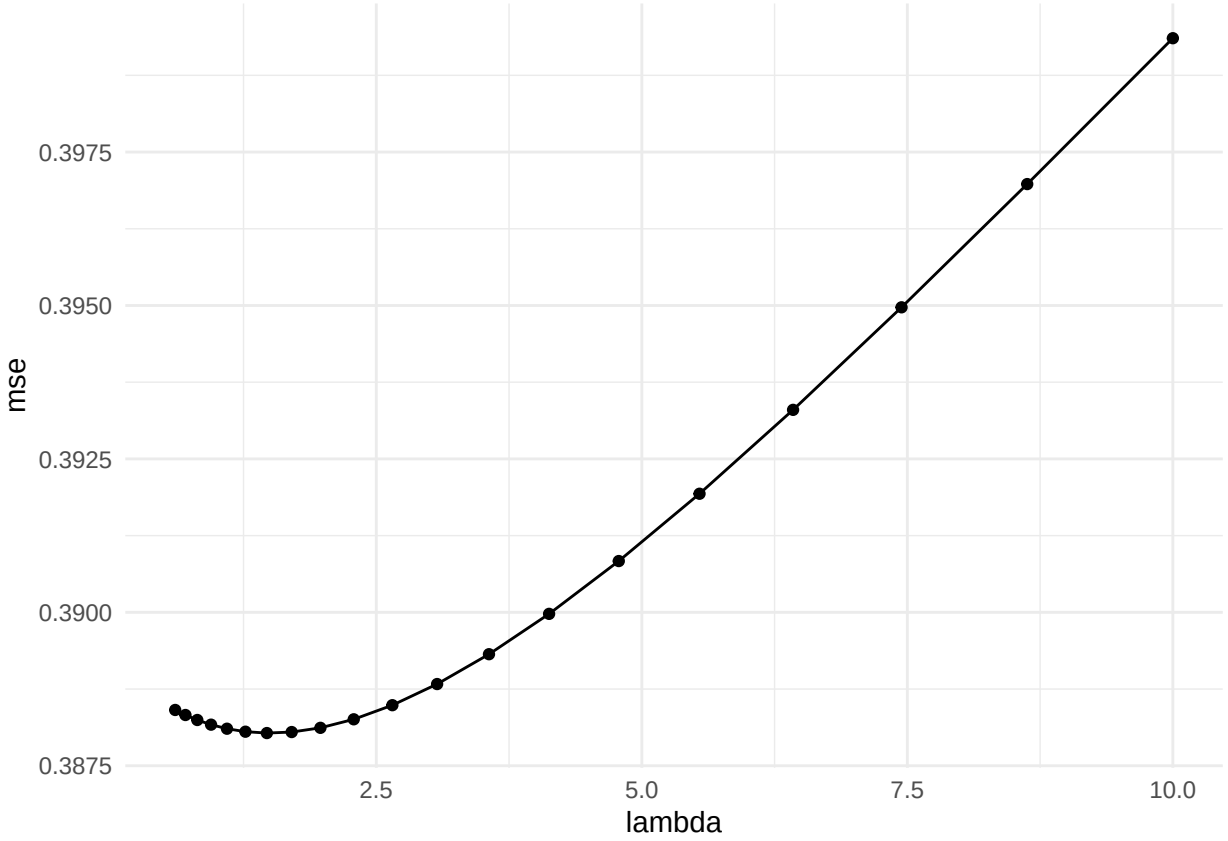
The following few lines now fits the model to training data for each combination of lambda in lambda_seq and cross-validation fold (model_fit column), computes predictions for the corresponding test data (predicted column), extracts the observed response variables from the test sets (actual column) and finally computes the mean squared error mse as the mean squared difference between actual and predicted

```
model_df <- cv_prostate %>%
    # One row for each combination of lambda and fold
    crossing(lambda = lambda_seq) %>%
    # Fit model to training data in each row
    mutate(model_fit = map2(train, lambda, ~lm.ridge(.x, formula_ridge, .y)),
           # Compute predicted valuea on test data
           predicted = map2(test, model_fit, ~predict.ridge(.x, .y, formula_ridge)),
           # Extract actual values from test data
           actual = map(test, ~(model.frame(formula_ridge, .x) %>%
```

```
                                    model.extract("response"))),
        # Compute mse
        mse = map2_dbl(predicted, actual, ~mean((.x - .y)^2)))
```

Finally, we plot the mean squared error as a function of hyperparameter by averaging over the folds for exach value of lambda

```
model_df %>%
    group_by(lambda) %>%
    summarise(mse = mean(mse)) %>%
    ggplot(aes(x = lambda, y = mse)) +
    geom_point() +
    geom_line()
```



Where the best hyperparameter λ is given by

```
best_lambda <- model_df %>% group_by(lambda) %>% summarise(mse = mean(mse)) %>% top_n(1, -mse) %>% pull
best_lambda
```

```
## [1] 1.469654
```

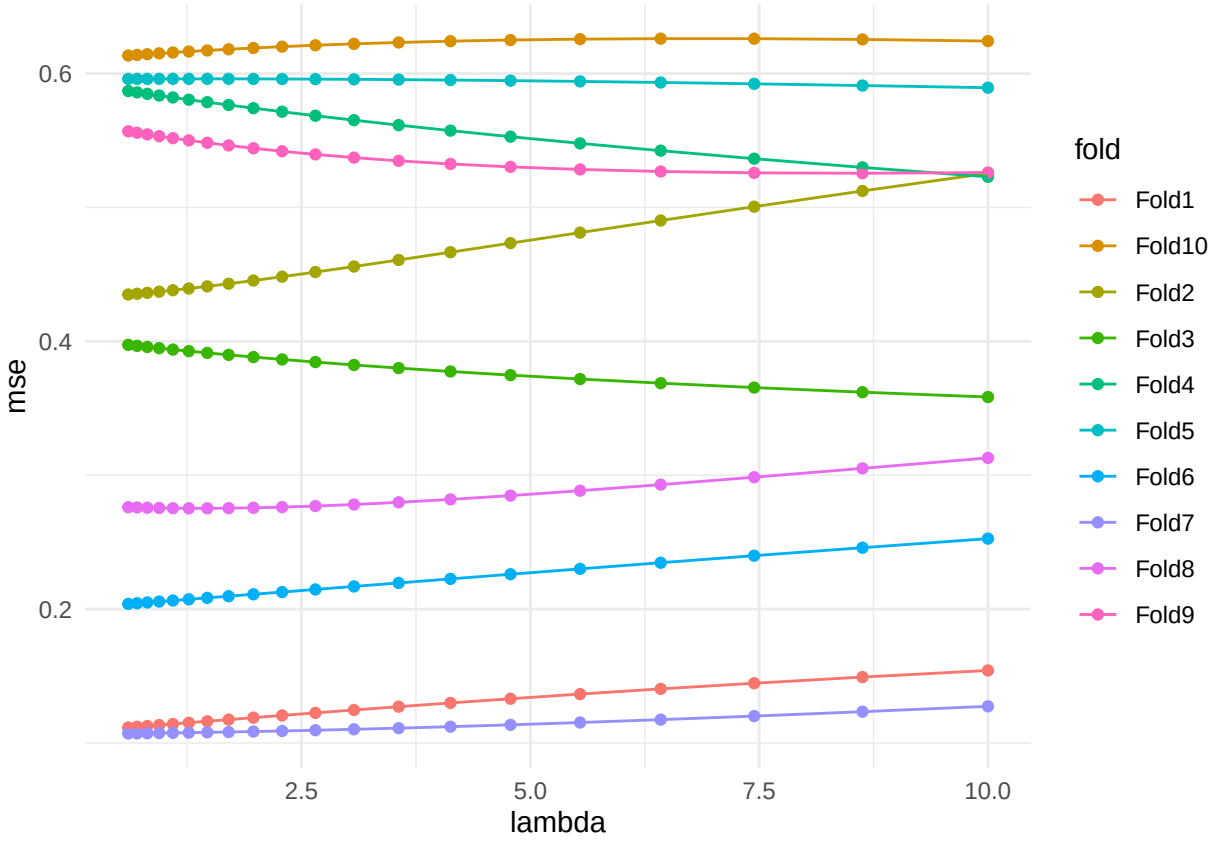
And, in consequence, the best ridge regression model is

```
best_ridge <- lm.ridge(prostate_prep_train, formula_ridge, best_lambda)
```

> Notice that in the project sheet given, the lambda vs mean squared error plot is completely diferent, decreasing almost until a value of lambda equals 6. In this case, the minimum is much smaller (around 1.5). As a consequence, cross-validation suggests much less shrinkage in this case than in the other one (even when we have the exact same data). In conclusion, the optimal lambda seems to be extremely sensitive to the random splitting of the folds.

Finally, we plot lambda against mse for each fold (instead of averaging).

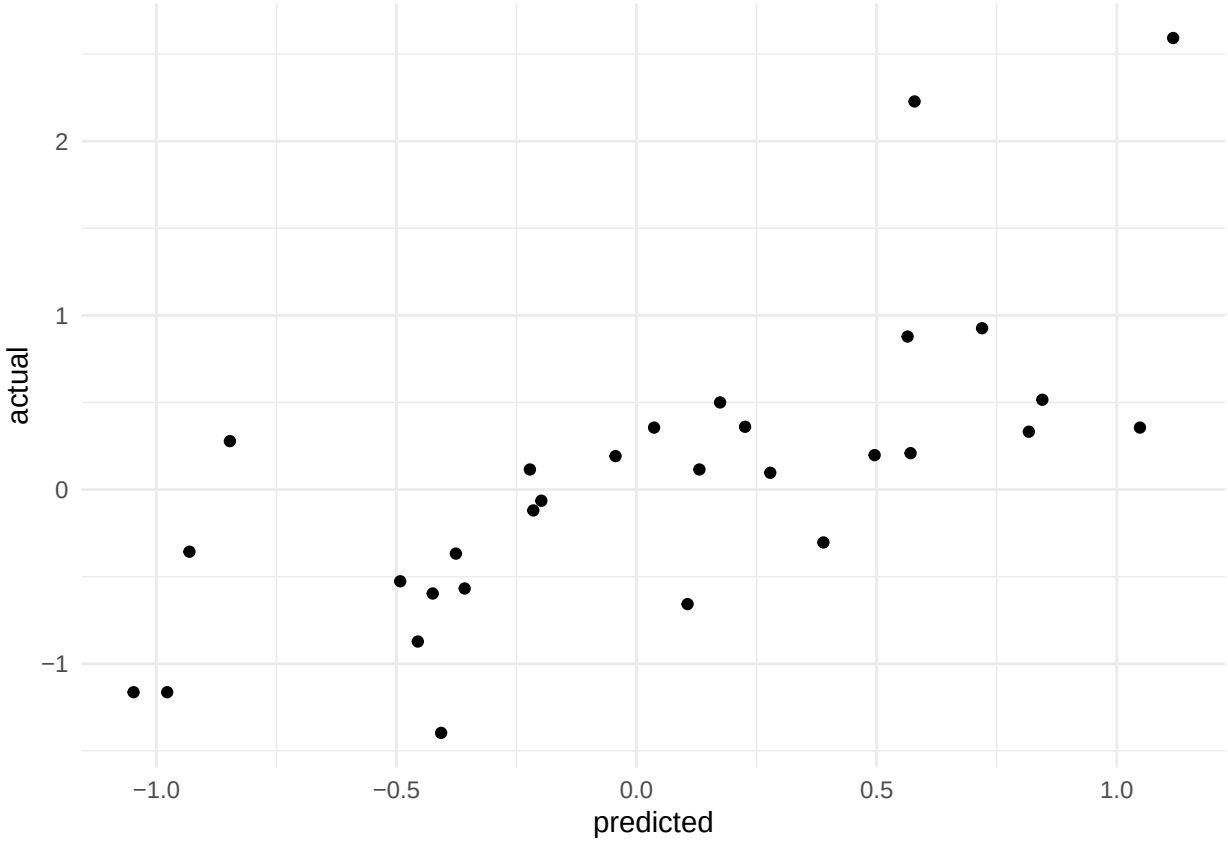
```
model_df %>%
    ggplot(aes(x = lambda, y = mse, group = fold)) +
    geom_point(aes(color=fold)) +
    geom_line(aes(color=fold))
```



The behaviour is extremely different between folds. For some of them, it appears that the mse is increasing as a function of lambda and for others it appears to be decreasing. Furthermore, for some folds the mean square error is bigger than .6 and for others almost 0. This is one of the arguments in favour of doing cross-validation.

We can finally plot the predicted against observed for the test data and compute test mse fo this choice of `lambda`

```
prostate_prep_test %>% mutate(predicted = predict.ridge(., best_ridge, formula_ridge),
                              actual = model.frame(formula_ridge, .) %>%
                                  model.extract("response")) %>%
    ggplot(aes(x = predicted, y = actual)) +
    geom_point()
```



Trees

Task 3

Find a (near) optimal value for `cp` *using cross-validation by applying techniques as in Taks 2 to the* `cv_prostate` *data.frame. Use all variables (*`formula = lpsa ~ .`*) rather than just* `lpsa` *above and compare test mean squared error with that of the ridge regression.*

The following functions were made to simplify notation in the calculation of cross-validation and lets us build the tree (for a custo dataframe and cp) and give the prediction

```
build.tree <- function(data, cp, formula){
  rpart(formula, data.frame(data), control = rpart.control(minsplit = 10, cp = cp))
}
predict.tree <- function(data, tree_fit){
  predict(tree_fit, data.frame(data))
}
```

Splitting the training data into 10 folds

```
n_fold <- 10
cv_prostate <- cv_fold(prostate_prep_train, n_fold)
```

We fit a model with all explanatory variables

```
formula_tree <- lpsa ~ .
```

With the following sequence of hyperparameters for cp

```
cp_seq <- exp(seq(0, log(1.5), length.out = 50)) -1
cp_seq
```

```
##  [1] 0.000000000 0.008309129 0.016687299 0.025135085 0.033653065 0.042241821
##  [7] 0.050901943 0.059634023 0.068438658 0.077316453 0.086268014 0.095293955
## [13] 0.104394894 0.113571453 0.122824262 0.132153954 0.141561167 0.151046546
## [19] 0.160610740 0.170254404 0.179978199 0.189782789 0.199668848 0.209637051
## [25] 0.219688081 0.229822627 0.240041382 0.250345045 0.260734323 0.271209927
## [31] 0.281772574 0.292422988 0.303161897 0.313990037 0.324908150 0.335916983
## [37] 0.347017289 0.358209829 0.369495370 0.380874683 0.392348549 0.403917753
## [43] 0.415583086 0.427345348 0.439205345 0.451163888 0.463221795 0.475379894
## [49] 0.487639016 0.500000000
```

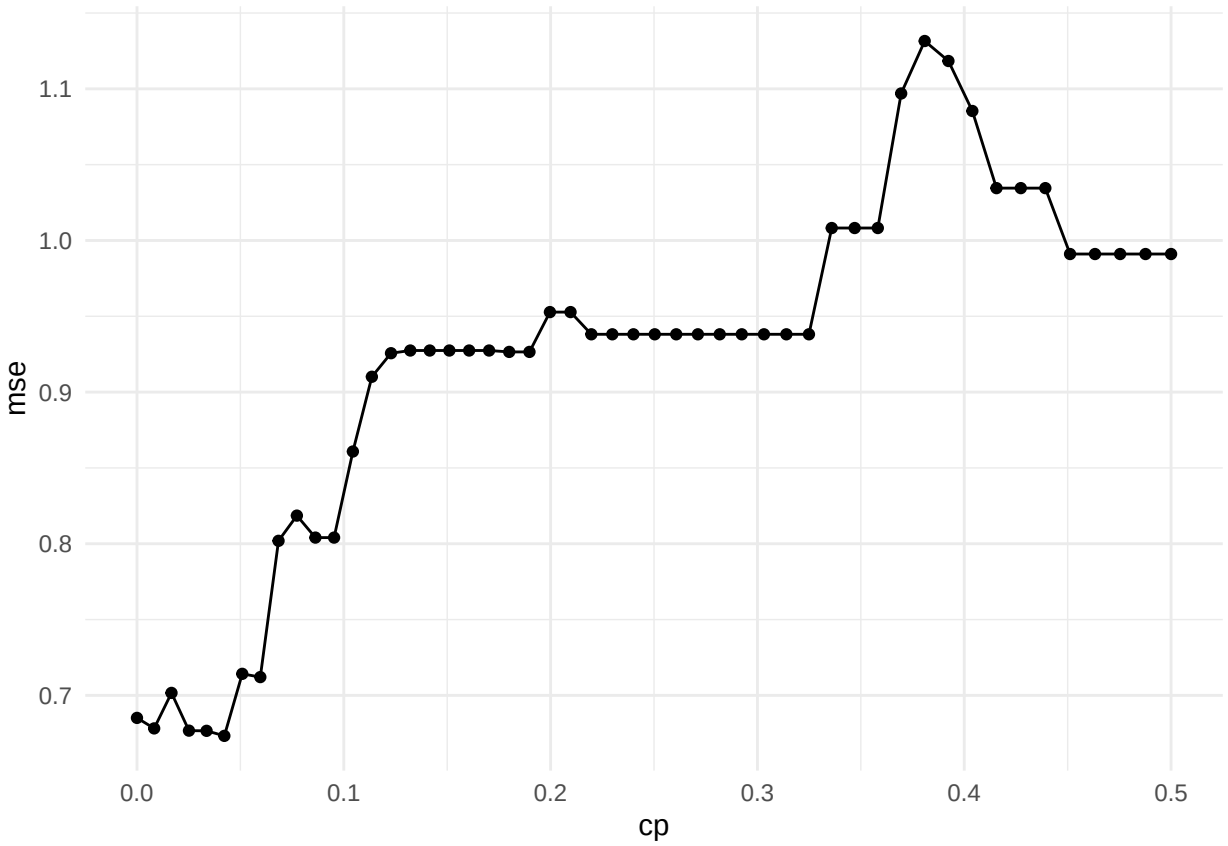
The following lines fit the model to training data for each combination of cp and cross-validation fold, computed the prediction and computes the mean squared error

```
model_df2 <- cv_prostate %>%
             # One row for each combination of cp and fold
             crossing(cp = cp_seq) %>%
             mutate(
                    # Train model with current fold and cp
                    model_fit  = map2(train, cp, ~build.tree(.x, .y, formula_tree)),
                    # Compute predicted valuea on test data
                    predicted = map2(test, model_fit, ~predict.tree(.x, .y)),
                    # Extract actual values from test data
                    actual = map(test, ~(model.frame(formula_tree, .x) %>%
                                         model.extract("response"))),
                    # Compute mse
                    mse = map2_dbl(predicted, actual, ~mean((.x - .y)^2))
             )
glimpse(model_df2)
```

```
## Observations: 500
## Variables: 8
## $ fold      <chr> "Fold1", "Fold1", "Fold1", "Fold1", "Fold1", "Fold1", "Fo...
## $ train     <list> [<tbl_df[60 x 9]>, <tbl_df[60 x 9]>, <tbl_df[60 x 9]>, <...
## $ test      <list> [<tbl_df[7 x 9]>, <tbl_df[7 x 9]>, <tbl_df[7 x 9]>, <tbl...
## $ cp        <dbl> 0.000000000, 0.008309129, 0.016687299, 0.025135085, 0.033...
## $ model_fit <list> [<3.00000000, 1.00000000, 7.00000000, 5.00000000, 6.0000...
## $ predicted <list> [<0.6872187, 1.5196182, -1.7143391, 1.5196182, -0.155512...
## $ actual    <list> [<1.3060484, 0.9758476, -0.6265292, 0.4642575, -0.103548...
## $ mse       <dbl> 0.4364823, 0.4364823, 0.3481956, 0.4682360, 0.4140499, 0....
```

We plot the mean squared error as a function of hyperparameter by averaging over the folds for exach value of cp

```
model_df2 %>%
  group_by(cp) %>%
  summarise(mse = mean(mse)) %>%
    ggplot(aes(x = cp, y = mse)) + geom_point() +
    geom_line()
```



Where the best hyperparameter 'cp' is given by

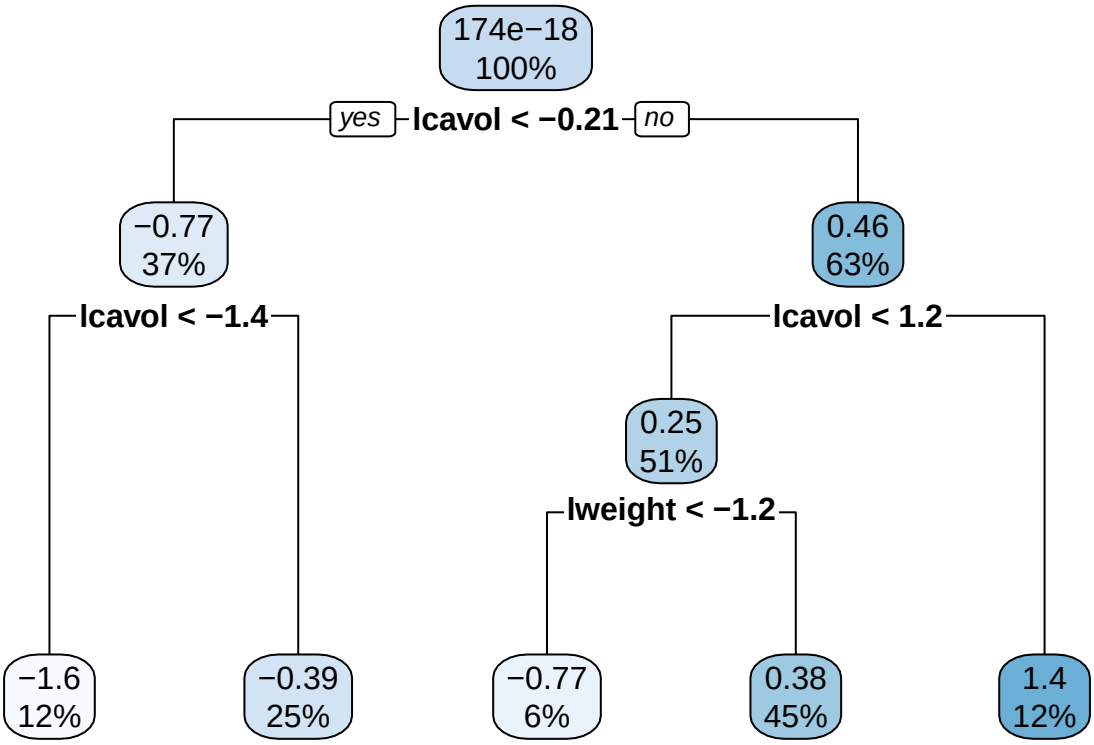
```
best_cp <- model_df2 %>% group_by(cp) %>% summarise(mse = mean(mse)) %>% top_n(1, -mse) %>% pull(cp)
best_cp
```

```
## [1] 0.04224182
```

And, in consequence, the best tree model is

```
best_tree <- build.tree(prostate_prep_train, best_cp, formula_tree)
rpart.plot(best_tree)
```

```
## Warning: Bad 'data' field in model 'call' (expected a data.frame or a matrix).
## To silence this warning:
##     Call rpart.plot with roundint=FALSE,
##     or rebuild the rpart model with model=TRUE.
```



We now compare the performance of the tree model agains the ridge regression in terms of the mean squared error using the test dataset

```
mse_ridge <- prostate_prep_test %>% mutate(predicted = predict.ridge(., best_ridge, formula_ridge),
                                actual = model.frame(formula_ridge, .) %>%
                                    model.extract("response")) %>%
    summarise(mse = mean((predicted - actual)^2))

mse_tree <- prostate_prep_test %>% mutate(predicted = predict.tree(., best_tree),
                                actual = model.frame(formula_tree, .) %>%
                                    model.extract("response")) %>%
    summarise(mse = mean((predicted - actual)^2))

mse_ridge
```

```
## # A tibble: 1 x 1
##     mse
##   <dbl>
## 1 0.349
```

```
mse_tree
```

```
## # A tibble: 1 x 1
##     mse
##   <dbl>
## 1 0.384
```

In this case, the mean squared error is slightly smaller for the ridge regression model.

Tree bias and variance

Simple decision trees are known for their high variance and small bias. In this part we will look further into this issue by a Monte-Carlo study. In particular, we will look at variance and bias of an estimator $\hat{f}(x)$ of $f(x)$ for various values of fixed x. We will do so by

- Choosing a (non-constant) function $f(x)$, $0 \leq x \leq 1$, distributions for the input variable X and the observation error ϵ, and a sample size n.
- Simulate N samples of size n from the distributions of X and $Y = f(X) + \epsilon$.
- For each of the N samples, fit $\hat{f}$, and compute its values on a grid.
- Approximate bias/variance for each value on the grid by averaging over the N samples.

Task 4

Pick your own function and distribution as above and repeat the analysis, fitting an unpruned decision tree using ***rpart*** *instead of a cubic (try a few values of* ***minsplit****). Do the trees balance variance and squared bias well?*

We'll estimate the following function

$$f(x) = \sin(x) + \cos(x^2) + sin(x)cos(x)$$

```
f <- function(x){
    sin(x) + cos(x^2) + sin(x)*cos(x)
}
```

And the following function to simulate a sample of size n (uniform X)

```
sim_data <- function(n = 100, f. = f, sd = 1/3){
    data.frame(x = runif(n)) %>% mutate(y = f(x) + rnorm(n, sd = sd))
}
```

A grid of points for which the performance will be evaluated

```
newdata <- data.frame(x = 0:50/50)
```

Number of Monte-Carlo samples

```
N <- 100
```

Finally, we plot the curves approximating bias, variance and mse of the tree regression applied to the function $f(x) = \sin(x) + \cos(x^2) + sin(x)cos(x)$ for several diferent values of misplit.

```
build.tree2 <- function(d, minsplit=10){
  rpart(y ~ ., data.frame(d), control = rpart.control(minsplit = minsplit))
}

tree_bias_variance_plot <- function(minsplit){
  tibble(data = rerun(N, sim_data())) %>% # Draw N samples

    mutate(
            # Fit tree model
            fit = map(data, ~build.tree2(.x, minsplit)),
            # Predict
            predicted = map(fit, ~mutate(newdata, predicted = predict(.x, newdata = newdata)))
    ) %>%
    unnest(predicted, .id = "name") %>%
    # Get aggregates
```

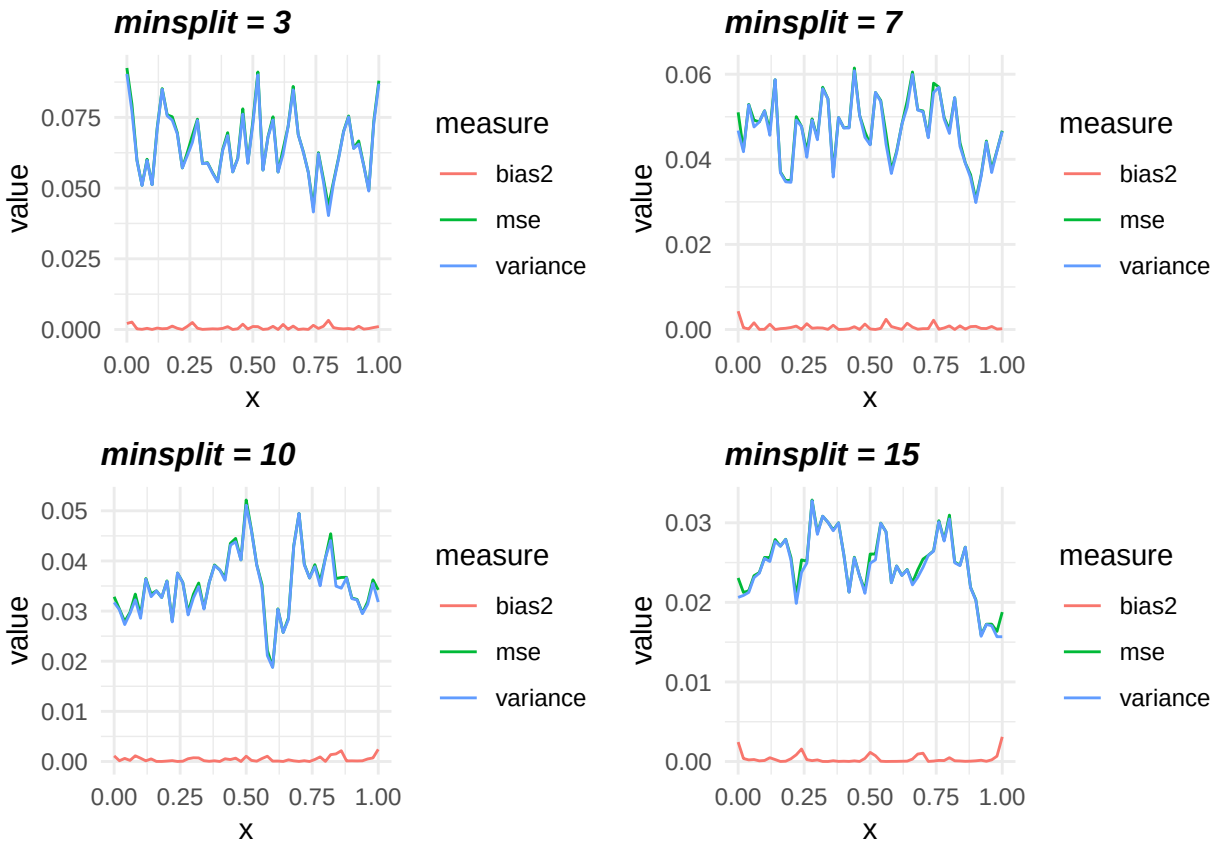
```
    group_by(x) %>%
    summarise(
              bias2 = mean(f(x) - predicted)^2,
              variance = var(predicted),
              mse = bias2 + variance
    ) %>%
    # Plot
    gather(
          key = "measure",
          value = "value",
          -x
    ) %>%
    ggplot(aes(x = x, y = value, color = measure)) +
    geom_line() +
    ggtitle(paste("minsplit =",minsplit)) +
    theme( plot.title = element_text(size=12, face="bold.italic") )
}

# To make plots with several diferent values of misplit.
p1 <- tree_bias_variance_plot(3)
```

```
## Warning: The `.id` argument of `unnest()` is deprecated as of tidyr 1.0.0.
## Manually create column of names instead.
## This warning is displayed once per session.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```

```
p2 <- tree_bias_variance_plot(7)
p3 <- tree_bias_variance_plot(10)
p4 <- tree_bias_variance_plot(15)

grid.arrange(p1, p2, p3, p4, nrow = 2)
```



As we can see from the plots, the squared bias and variance doesn't seem to be fairly well balanced. It always appear to show a really high variance and a small bias.

Task 5

Bagging can be used to reduce variance of trees. It works by averaging trees applied to boostrap samples of data. Write a function that takes ***data*** *as input, draws* `B` *Bootstrap resamples (draw with replacement) from data, fits a decision tree (using* ***rpart****) to each resample and predicts values at* ***newdata****. The function should then return a* ***data.frame*** *with columns* ***x*** *and* ***predicted****, where* ***predicted*** *is the average of the resampled predictions. Finally, repeat task 4 with this new function*

The following function takes data as input, draws `B` Bootstrap resamples, fits a decision tree to each resample and predicts values to newdata.

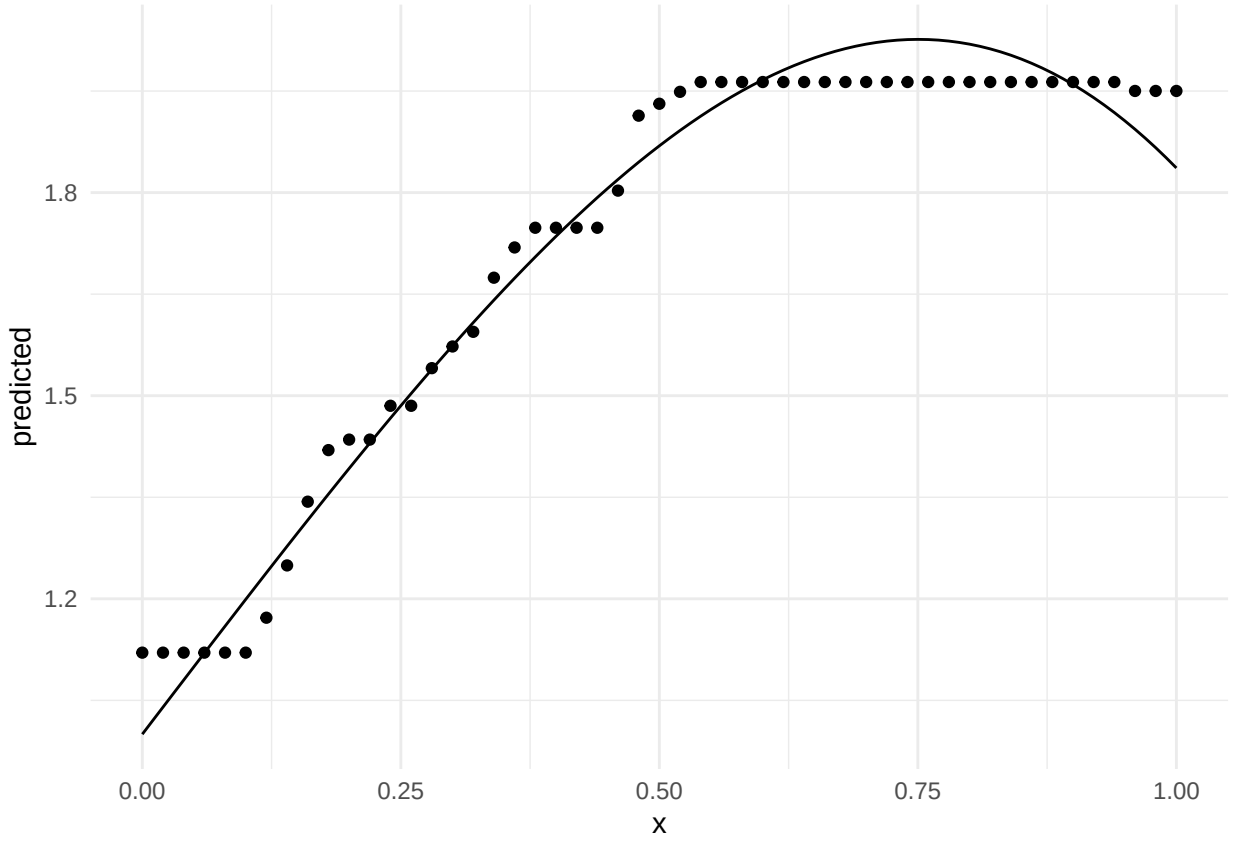
```
bag_tree <- function(data, newdata = data.frame(x = 0:50/50), B = 10){

  bootstrap_samples <- tibble(n_sample = 1:B) %>% #Number of samples
                       mutate(
                           # Sample with replacement from data
                           data = map(n_sample, ~sample_n(data, size=nrow(data), replace=TRUE)),
                           # Fit a tree model to each sample
                           fit = map(data, ~build.tree2(.x) ),
                           #Predict with each tree model the values for newdata
                           predicted = map(fit, ~mutate(newdata, predicted = predict(.x, newdata = newd
                       )
    # Return the average for of the resampled predictions
    unnest(bootstrap_samples, predicted) %>%
              group_by(x) %>%
```

```
            summarise(predicted = mean(predicted))
}
```

As an exaple, we can plot the predicted values with out `bag_tree`function as follows

```
newdata <- data.frame(x = 0:50/50)
data <- data.frame(x = newdata) %>%
           mutate(y = f(x))
bag_tree(data,newdata) %>%
      ggplot() +
      geom_point(aes(x = x, y = predicted)) +
      stat_function(fun = f)
```



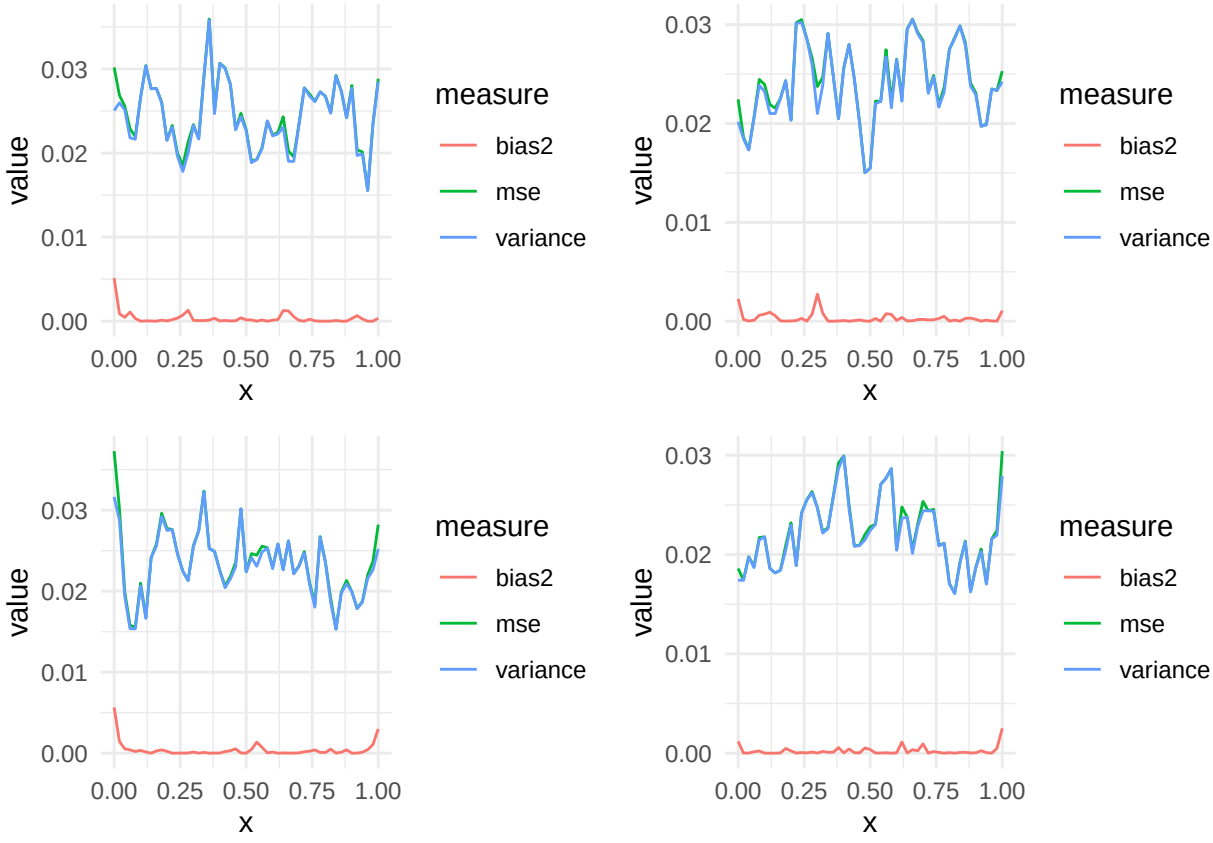
Finally, as in task 4, we plot the curves approximating bias, variance and mse of the averaged tree regression applied to the function $f(x) = \sin(x) + \cos(x^2) + sin(x) * cos(x)$ for several diferent values of misplit.

```
N <- 100
tree_bias_variance_plot2 <- function(minsplit){
  tibble(data = rerun(N, sim_data())) %>%
        mutate(
          predicted = map(data, ~bag_tree(.x))
        ) %>%
        unnest(predicted, .id = "name") %>%
        group_by(x) %>%
        summarise(
                  bias2 = mean(f(x) - predicted)^2,
                  variance = var(predicted),
                  mse = bias2 + variance
```

```
        ) %>%
        gather(
              key = "measure",
              value = "value",
              -x
        ) %>%
        ggplot( aes(x = x, y = value, color = measure) ) +
        geom_line()
}

p1 <- tree_bias_variance_plot2(3)
p2 <- tree_bias_variance_plot2(7)
p3 <- tree_bias_variance_plot2(10)
p4 <- tree_bias_variance_plot2(15)

grid.arrange(p1, p2, p3, p4, nrow = 2)
```



We notice that the variance is considerably smaller in every case (when comparing with the non-averaged trees) concluding that bagging can be used to succesflly reduce variance of trees.
